

# An Introduction to R

Jamie Monogan  
The University of North Carolina at Chapel Hill\*  
The Odum Institute for Research in Social Science

## Objectives

By the end of this course, participants will be able to:

- Use resources about R to look-up more information on it.
- Load, manipulate, and use data in R.
- Run basic models.
- Perform matrix algebra in R.
- Explain the logic of programming in R.
- Program maximum likelihood estimators.
- Install packages and use them for upper-level methods.
- Draw graphs using the `base` and `lattice` graphics packages.

## 1 Background

### 1.1 Who am I? Who are you? How can Odum serve you?

- Sign-in
- Office hours
- Short courses

### 1.2 What is R?

R is a platform for the object-oriented statistical programming language S. It is widely used in statistics and has become quite popular in political science over the last decade. R, which is shareware, is similar to S-plus, which is the commercial platform for S. Essentially R can be used as either a matrix-based programming language or as a standard statistical package that operates much like Stata, SAS, or SPSS.

---

\*These notes draw heavily from a short course Luke Keele did at Oxford University on May 21, 2004 and another course by Evan Parker-Stephen for UNC's political science department on November 2, 2006. My thanks to Luke & Evan for sharing this information. This document was last revised on May 27, 2009.

### 1.3 Where Can I get R?

The beauty of R is that it is shareware, so it is free to anyone. To obtain R for Windows, Mac, or Linux go to The Comprehensive R Archive Network (CRAN) at <http://www.r-project.org/>. Just download the executable file, and it will install itself.<sup>1</sup> Unfortunately, there are not update patches, so you must completely install a new version whenever you would like to upgrade.

## 2 Resources

As questions emerge when you use R, here are a few resources you may want to consider:

- Within R, there is the “Introduction to R,” which is under the “Help” pull down menu under “Manuals,” as well as the `?` and `help` commands. Also note the html-based help search.
- To do a web-based search, Rseek (powered by Google) is the only worthwhile search engine. <http://www.rseek.org/>
- Monogan’s R website, <http://www.unc.edu/~monogan/computing/r/> has a number of links and resources posted on it. Feel free also to come to Monogan’s consulting hours in the Odum Institute, which are also posted on the webpage.
- Jeff Harden’s webpage also includes many resources. <http://www.unc.edu/~jjharden/R.html>
- John Fox’s *Introduction to Regression in R and S-plus*.
- For a more in-depth treatment see Venables and Ripley’s *Modern Applied Statistics in S*.

## 3 Getting Started

Once you have installed R, there will be an icon on your desktop. Double click it and R will start up. You will notice the R does have a few pull-down menus, but mostly commands in R are entered on the command line:

```
>
```

Some preliminaries on entering commands.

- Expressions and commands in R are case-sensitive.
- Command lines do not need to be separated by any special character like a semicolon as in Limdep, SAS, or Gauss.
- Anything following the pound character (`#`) R ignores as a comment.
- An object name must start with an alphabetical character, but may contain numeric characters thereafter. A period may also form part of the name of an object. For example, `x.1` is a valid name for an object in R.
- You can use the arrow keys on the keyboard to scroll back to previous commands.

---

<sup>1</sup>Linux users will find it easier to install from a terminal. Terminal code is available at <http://www.unc.edu/~monogan/computing/linux/>.

**Saving Output.** Your output from a session in R can be saved using the `sink` command. To save your session to the file “Rintro.txt”:

```
> sink('D:/temp/Rintro.txt')
```

Notice that the path directory uses forward slashes rather than backslashes. This is different from normal Windows syntax.

Now that we have created an output file, if you use the `print` command, your output will be saved to “Rintro.txt”. You can print strings of text like this:

```
> print('The mean of variable x is...')
```

and your “D:\temp\Rintro.txt” file will contain:

```
[1] 'The mean of variable x is...'
```

Another useful printing command is the `cat` command since it lets you mix objects in R with text. For example, let’s say we create the variable `x`:

```
x <- rnorm(1000)
```

By way of quick explanation: this syntax draws randomly 1,000 times from a standard normal distribution and assigns the values to the vector `x`. Observe the arrow (`<-`), which is R’s assignment operator. Now we can print the mean of these 1,000 draws as follows:

```
> cat("The mean of variable x is...", mean(x), '\n')
```

So now objects from R can be embedded into the statement you print. The character `\n` puts in a carriage return. You can also print any statistical output using the either `print` or `cat` commands. Remember, though, your output does not go to the log file unless you use one of the `print` commands. You can also copy and paste into Word or a text editor out of the R window. To turn off the `sink` command:

```
> sink()
```

**Objects.** R saves any object you create. To list the objects you have created in a session use either of the following commands:

```
> objects()
```

```
> ls()
```

To remove all the objects in R type:

```
> rm(list=ls(all=TRUE))
```

**Quitting.** To quit R type:

```
> q()
```

**Packages.** R has many useful add on components that are called packages. Later, we will use

some packages to perform advanced methods not available in most econometric software. To load a package you simply type:

```
> library(packagename)
```

**Text Editing.** While one can type R commands one line at a time directly into the R console this is cumbersome and not at all efficient for writing programs. So instead most users type R commands into a text editor and then copy and paste them into R. Most simply this is done with the notepad. Type your R commands into the notepad and then cut and paste them into R. You can also use more advanced setups with text editors like Emacs or WinEdt.

**Reading in Data.** Getting data into R is quite easy. There are two primary ways to import data. The first is to read in a delimited text file with the `read.table` command. R will read in a variety of delimited files. (For all the options associated with this command type `?read.table` in R.) As an example read in the following dataset by Poe and Tate (1994) called `hmnrights.txt`:

```
> hmnrights <- read.table("D:/temp/hmnrights.txt", header=TRUE, na="NA")
```

Remember you will have to specify the location of the file yourself depending on where you have it saved. The header option specifies whether the first line of the file contains the names of the variables, which here are: (1) **country**, (2) **democ**, a scale rating the level of democracy in a country, (3) **sdnew**, U.S. State Department scale of Political Terror, (4) **military**, a dummy variable for a military regime, (5) **gnpcats**, level of gnp in four categories, (6) **lpop**, log of population, (7) **civ.war**, a dummy variable for whether a country was involved in a civil war, and (7) **int.war**, a dummy variable for whether a country was involved in an international war. All variables are for 1993 only. Now type:

```
> hmnrights
```

Our new dataset will print to the screen. This is not recommended with large datasets. To check the names of the variables in our dataset type:

```
> names(hmnrights)
```

You also can import data from another many other statistical packages. The `foreign` package in R makes it very easy to bring in data from other statistical packages, such as SAS, SPSS, and Stata. To install the `foreign` package (or any other R package) type:

```
> install.packages("foreign")
```

Then follow the mirror-selection prompts. After this, to bring in a dataset from Stata type:

```
> library(foreign)
```

```
> data.name <- read.dta(file.choose())
```

which lets you browse for a Stata dataset. Any data in Stata format that you select will be converted to R format. To read data in directly replace `file.choose()` with the path to the file and the file name like so:

```
> hmnrights <- read.dta('D:/temp/hmnrights.dta')
```

One word of warning, the value labels from Stata do not always translate neatly. Often it is not a

problem, but if you are having problems, try importing the data without value labels. As a quick check on whether the data imported correctly, try the command:

```
> fix(hmnrghs)
```

Please note: before continuing with R commands, you must close the data editor window. (Be careful that you are only closing the editor window.)

**Missing Data.** R designates missing values with `NA`. It translates missing values from other statistics packages into the `NA` missing format. To create a new data set with all the missing values deleted through listwise deletion, type:

```
> hmnrghs <- na.omit(hmnrghs)
```

**Arrays, Vectors, Data Frames, and Matrices.** R distinguishes between vectors, data frames, and matrices. Vectors are indexed by length and matrices are indexed by rows and columns. Data frames are a matrix that R designates as a data set. With a data frame, the columns of the matrix can be referred to as variables. After reading in a data set, R will treat your data as a data frame, which means that you can refer to any variable within a data frame by adding `$VARIABLENAME` to the name of the data frame. For example, we can print out the variable `country` in order to see which countries are in the dataset:

```
> hmnrghs$country
```

Or you can use the `attach` command, which lets so you use variable names individually. For example, after typing

```
> attach(hmnrghs)
```

you can now refer to `country` as an individual vector, without having to refer to the name of the data frame where it is located. Importantly, the “detach” command works the same way:

```
> detach(hmnrghs)
```

## 4 Logical Statements

Logical statements in R are evaluated as to whether they are `TRUE` or `FALSE`. Table 1 summarizes the different logical operators in R.

Table 1: Logical Operators in R

Operator	Means
<code>&lt;</code>	Less Than
<code>&lt;=</code>	Less Than or Equal To
<code>&gt;</code>	Greater Than
<code>&gt;=</code>	Greater Than or Equal To
<code>==</code>	Equal To
<code>!=</code>	Not Equal To
<code>&amp;</code>	And
<code> </code>	Or

For example, suppose we wanted to know which countries have had a civil war and have above

average GNP:

```
> wealthWar <- hmnrghts$civ_war==1 & hmnrghts$gnpcats>2.6
```

returns a vector of TRUE and FALSE for every observation. In this case, we see that there is only one country with above average GNP that have been involved in a civil war (in 1992 anyway).

## 5 Recoding

Often we need to recode variables, and there are a variety of ways to do this in R. The basic syntax for creating mathematical transformations of variables follows the form of the example below. Suppose we want the actual population of each country instead of its logarithm:

```
> hmnrghts$pop <- exp(hmnrghts$lpop)
```

Any type of mathematical operator could be substituted in for `exp`. Another standard type of recoding we might want to do is the creation of a dummy variable that is coded as 1 if the observation meets certain conditions and 0 otherwise. For example, suppose instead of having categories of GNP, we just want to compare the highest category of GNP to all the others:

```
> hmnrghts$gnp.dummy <- as.numeric(hmnrghts$gnpcats>3)
```

Here we use a logical statement and modify it with the `as.numeric` statement which turns each TRUE into a 1 and each FALSE into a 0. Let's say instead of having the population of each country we wanted a 3 category ordinal level variable.

```
> library(car)
```

```
> hmnrghts$pop.3<-recode(hmnrghts$lpop, 'lo:15.18=1;15.18:17.11=2;17.11:hi=3')
```

In commands above, we have created a new variable from 1 to 3, where countries with a log population below 15 are coded as 1 and so on. Of course, this syntax could be used to create dummy variables, ordinal variables, or a variety of other recoded variables.

There is one quirk in R when it comes to recoding variables or creating new ones. If you recode or create new variables for an attached data set, those changes are made to global variables and not to the specific data frame that you would like to change. To make changes to a specific data set make certain it is detached before making the changes and use `$` syntax to refer to variables in a data frame. If you like, re-**attach** after the changes are completed.

**Factors.** Categorical variables in R can be given a special designation as factors. If you designate a categorical variable as a factor, R will treat it as such in statistical operation and create dummy variables for each level when it is used in a regression. If you import a variable with no numeric coding, R will automatically treat the variable as a factor. For example, the variable `country` in the data set is automatically treated as a factor. Once R thinks a variable is a factor it has a series of special commands that can be used. For example, try:

```
> levels(country)
```

This lists all the levels in the factor with their names. The levels of a factor need not be named. To change the levels of a factor to character strings, it exactly the same as recoding the variable in the second example, but use quotes around the character string that you want the level of a factor to take on. To change which level is the first level (i.e. to change which category R will use as the reference category in a regression) use the `relevel` command. The following code establishes the variable “country” as a factor, and then sets “united states” as the reference category:

```
> hmnrghts$country <- as.factor(hmnrghts$country)
> hmnrghts$country <- relevel(hmnrghts$country, "united states")
```

## 6 Basic Models

### 6.1 Ordinary Least Squares

Estimating an OLS model in R is done with the `lm` command. Let's predict human rights violations as function of the variables `democ` and `gnpcats`:

```
> hmnrghts.model <- lm(sdnew ~ democ + gnpcats, data=hmnrghs)
```

`hmnrghs.model` is the name we have given to our regression model. In the linear model (`lm`) command itself, the dependent variable comes first, followed by a `~`, and all independent variables must be separated with a `+`. You also need to specify the data frame which R should use when it estimates the model. After typing the above command, R will not automatically print any output. To see the output you have to type the following command:

```
> summary(hmnrghts.model)
```

Which will give you the following output:

```
Call:
lm(formula = sdnew ~ democ + gnpcats)

Residuals:
Min       1Q   Median       3Q      Max
-2.4881   -0.5839   -0.1624    0.6421    3.0252

Coefficients:
              Estimate Std. Error t value Pr(> |t|)
(Intercept)   3.90950    0.17403   22.464 <2e-16 ***
democ         -0.09577    0.02370   -4.040  9.37e-05 ***
gnpcats       -0.32564    0.05879   -5.539  1.78e-07 ***
---
Signif codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error:  1 on 122 degrees of freedom
Multiple R-squared:  0.4105, Adjusted R-squared:  0.4009
F-statistic:  42.48 on 2 and 122 DF, p-value:  9.969e-15
```

It is now worth noting that R is an object-oriented environment. Whenever we run a model or create a data frame, we create an object, which consists of several components. Hence, we can call on these components of the `hmnrghs.model` object to obtain several other regression outputs including the `coefficients`, `residuals`, `cov.unscaled` (the variance-covariance matrix) and `fitted.values`.<sup>2</sup> To see these outputs take the model name and use the `$` to append the object. For example, to see only the coefficients from the model type:

---

<sup>2</sup>See section 16 on page 22 for examples of the usefulness of this.

```
> hmnrghts.model$coefficients
```

```
(Intercept)    democ    gnpcats  
3.90950        -0.09577   -0.32564
```

As far as model specification, the `lm` command is also designed to allow you to do interactions on the fly. For example, to test for an interaction between democracy and the level of GNP we would use the following command:

```
> hmnrghts.model2 <- lm(sdnew ~ democ + gnpcats + democ*gnpcats, data=hmnrghs)  
Type this command and then look at the results. You will see that R creates an interaction and includes it in the model. Lastly, if you wanted to run a model without a constant:  
> hmnrghts.model3 <- lm(sdnew ~ democ + gnpcats + democ*gnpcats-1, data=hmnrghs)
```

### 6.1.1 Output to L<sup>A</sup>T<sub>E</sub>X

For users of L<sup>A</sup>T<sub>E</sub>X or HTML, R can produce ready-to-use, error-free tables by installing the `xtable` package. To create a table for L<sup>A</sup>T<sub>E</sub>X, I enter the following syntax:

```
> install.packages('xtable')  
> library(xtable)  
> xtable(hmnrghts.model)
```

Which gives this output:

```
% latex table generated in R 2.6.2 by xtable 1.5-2 package  
% Tue Jun 10 00:06:33 2008  
\begin{table}[ht]  
\begin{center}  
\begin{tabular}{rrrrr}  
  \hline  
  & Estimate & Std. Error & t value & Pr(>|t|) \\  
  \hline  
(Intercept) & 3.9095 & 0.1740 & 22.46 & 0.0000 \\  
  democ & -$0.0958 & 0.0237 & -$4.04 & 0.0001 \\  
  gnpcats & -$0.3256 & 0.0588 & -$5.54 & 0.0000 \\  
  \hline  
\end{tabular}  
\end{center}  
\end{table}
```

Pasting this output into L<sup>A</sup>T<sub>E</sub>X, with one additional line for a caption, produces Table 2:

Table 2: Regression Model of Human Rights Violations

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	3.9095	0.1740	22.46	0.0000
democ	-0.0958	0.0237	-4.04	0.0001
gnpcats	-0.3256	0.0588	-5.54	0.0000

## 6.2 General Linear Models

The `glm` command allows you to run a variety of models. To use the `glm` command you must specify your formula, the family, and the link function. For example, let's say we wanted to predict whether a country has a military regime as a function of GNP and population using a logit model:

```
> mil.model <- glm(military ~ lpop + gnpcats,  
+ family=binomial(link=logit), data=hmnrghts)  
> summary(mil.model)
```

Which will give you the following output:

```
Call:  
glm(military ~ lpop + gnpcats, family=binomial(link=logit))  
  
Deviance Residuals:  
    Min       1Q   Median       3Q      Max  
-0.9886  -0.7021  -0.2784  -0.1170   3.0956  
  
Coefficients:  
            Estimate Std. Error t value Pr(> |t|)  
(Intercept)   2.7284     3.1318   0.871   0.3836  
democ         -0.1676     0.1907  -0.879   0.3794  
gnpcats       -0.9873     0.3509  -2.813   0.0049 **  
—  
Signif codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1  
  
Dispersion parameter for binomial family taken to be 1)  
Null deviance: 103.040 on 124 degrees of freedom  
Residual deviance: 84.893 on 122 degrees of freedom  
AIC: 90.893
```

```
Number of Fisher Scoring Iterations: 6
```

If you wanted to estimate a probit, you would use the exact same command, but substitute `link=probit` in the specification of the model. If you wanted to run a count model:

```
> democ.model<-glm(democ~military+civ_war,  
+ family=poisson(link=log), data=hmnrghts)
```

## 7 Probability Distributions

R allows you to use a wide variety of distributions for four purposes. For each distribution, R allows you to call the cumulative distribution function (cdf), probability density function (pdf), quantile function, and random draws from the distribution. All probability distribution commands consist of a prefix and a suffix. Table 3 presents the four prefixes, and their usage, as well as the suffixes for some commonly-used probability distributions.

If you wanted to know the probability that a standard normal observation will be less than 1.645, use the command `pnorm`:

Table 3: Using Probability Distributions in R

Prefix	Usage	Suffix	Distribution
p	cumulative distribution function	norm	normal
d	probability density function	logis	logistic
q	quantile function	t	$t$
r	random draw from distribution	f	$F$
		unif	uniform
		pois	poisson
		exp	exponential
		chisq	chi-squared
		binom	binomial

```
> pnorm(1.645)
```

Let's say you want to draw a scalar from the standard normal distribution: to draw  $\mathbf{a} \sim \mathcal{N}(0, 1)$ , use the command `rnorm`:

```
> a <- rnorm(1)
```

To draw a vector with 10 values from a  $\chi^2$  distribution with four degrees of freedom:

```
> c <- rchisq(10,4)
```

## 8 Vector and Matrix Algebra

**Assignment** The way to assign values to a vector or matrix is to use the `<-` command. So to create a vector `a` with the specific values of 3, 4, and 5:

```
>a <- c(3,4,5)
```

Or to create a 10 by 1 vector of 1's:

```
> a1 <- rep(1, 10)
```

To create a vector that contains values counting from 1 to 10, type:

```
> d <- c(1:10)
```

A more general command is the `seq` command, which allows you to define the intervals of a sequence, as well as starting and ending values. For example to create a sequence from -2 to 1 in increments of .25:

```
> e <- seq(-2,1, by=0.25)
[1] -2.00 -1.75 -1.50 -1.25 -1.00 -0.75 -0.50 -0.25 0.00 0.25 0.50 0.75
[13] 1.00
```

To create a 4 by 4 matrix **b** with values of 3:

```
> b <- matrix(3, ncol=4,nrow=4)
```

To draw a 10 by 10 matrix **b1** from  $N(10,4)$ :

```
> b1 <- matrix(rnorm(100, mean=10, sd=2), nrow=10, ncol=10)
```

To create a 2 by 2 matrix **X** (which we will use in a later example) in which you list the value of each cell column-by-column:

```
> X <- matrix(c(1,2,3,4),ncol=2,nrow=2) #fills-in by columns
```

Or, if you wanted to list the cell elements row-by-row in matrix **Z**:

```
> Z <- matrix(c(1,2,3,4),ncol=2,nrow=2,byrow=TRUE) #fills-in by rows
```

**Concatenation.** You can concatenate by column or by row with `cbind` and `rbind` commands. Suppose **f** and **g** are both  $n \times 1$  vectors:

```
> f <-c(3,4,5)
```

```
> g <-c(10,11,12)
```

```
> H <- cbind(f,g)
```

```
      f  g
[1,]  3 10
[2,]  4 11
[3,]  5 12
```

```
> L <- rbind(f,g)
```

```
      [1,] [2,] [3,]
f       3   4   5
g      10  11  12
```

**Subscripting** To call a specific value, you can index a vector **f** by the  $n$ th element, use `f[n]`. To index an  $n \times k$  matrix **H** for the value  $H_{ij}$ , where  $i$  represents the row and  $j$  represents the column, use the syntax `H[i,j]`. If you want to select all values of the  $j$ th column of **H** you can use `H[,j]`. For example, to return the second column of matrix **H**, type:

```
> H[,2]
```

To get the third row of **H**:

```
> H[2,]
```

**Vector Commands.** To take the sum of a vector:

```
> sum(a)
```

To take the mean of a vector:

```
> mean(a)
```

To take the variance of a vector:

```
> var(a)
```

Or take the standard deviation with the `sd` command.

### Matrix Algebra

In addition to creating vectors and matrices, the assignment command is also used when performing basic vector and matrix operations. For example, to assign vector `a` the sum of two vectors `b` and `c`, we type:

```
> a <- b + c
```

All other arithmetic operations, such as `+`, `-`, `*`, `/`, `^` (exponents), and `log` work similarly. Note that simple multiplication by `*` performs multiplication element by element, while `%*%` performs matrix multiplication. Some other useful matrix functions include:

To take the transpose of `b`:

```
> t(b)
```

To take the inverse of `b`:

```
> solve(b)
```

To obtain the length of a vector `a`:

```
> length(a)
```

To obtain the dimension of a matrix:

```
> dim(b)
```

Can you find the dimensions of `hmnrights`? The dimension should be the number of variables by the number of cases.

To illustrate the various matrix algebra operations R has available, let us work an applied example by computing the OLS estimator ( $\hat{\beta} = (X'X)^{-1}X'y$ ):

```
> y <- c(5,6)
> beta <- solve(t(X)%*%X)%*%t(X)%*%y
```

**Sampling** If you want to sample from a given object use the command `sample`. Suppose we want a sample of 10 ten numbers from `b1`, our 10 by 10 matrix of random numbers:

```
> b1 <- matrix(rnorm(100, mean=10, sd=2), nrow=10, ncol=10)
> s <- sample(b1,10)
```

This gives you a vector of ten random elements from `b`. However this is not the best way to perform a bootstrap. If you are interested in bootstrapping, install the `boot` package as below and run some examples:

```
> library(boot)
> example(boot)
```

**Apply.** The `apply` command is often the most efficient way to do vectorized calculations. For example, to calculate the means for all the variables in the `humanrights` data set:

```
> apply(as.matrix(hmnrghts[,3:7]),2,mean)
```

In the command here, we refer to columns 3 to 7 of the `hmnrghts` data frame, which are variables 3 to 7. The 2 tells R to apply a mathematical function along the columns of the matrix, and `mean` is the function we want to apply to each column. If we used a 1 instead of a 2, R would apply the calculation to the rows of the data frame. Any function defined in R can be used with the `apply` command.

Can you calculate the standard error of these variables using `apply`? One thing to note: If the data you import from Stata has value labels in it, the `apply` function will not work. It only works on variables that R has designated as a numeric object, which is why here I use the `as.numeric` command to temporarily coerce the `hmnrghts` data frame into a matrix.

## 9 Loops

Loops are easy to write in R and can be used to repeat calculations. The basic structure for a loop is:

```
> for (i in 1:10) {COMMANDS}
```

To demonstrate how loops work, let's do a demonstration of the law of large numbers:

```
# First create a storage matrix
store <- matrix(NA,1000,1)
# Start the Loop
for (i in 1:1000){
a <- rnorm(i)
store[i] <- mean(a)
}
# Plot the Output
plot(store, type='o')
```

In this simple program with a loop, I demonstrate the Law of Large Numbers. In each pass of the program, R samples from a  $\mathcal{N}(0,1)$  distribution and then takes the mean of that sample. The sample size increases with each pass of the loop. As such, our plot should show the mean of

the sample converging to the true mean of zero as the sample size increases. (More on the `plot` command this afternoon.)

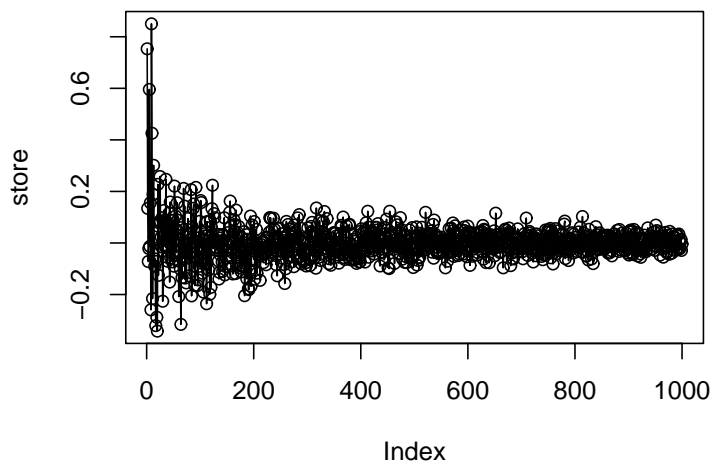


Figure 1: The Law of Large Numbers and Loops in Action

In Figure 1 you can see the results for this simple program. Loops are necessary for many types of programs, particularly if you want to do a Monte Carlo analysis. However, they should be avoided whenever possible, since they will slow down your program considerably. If possible try to use the `apply` command. R also supports while loops which follow a similar command structure:

```
j <- 1
while(j < 10) {COMMANDS
j <- j + 1
}
```

## 10 Functions

Another way to program in R is to create your own functions with the `function` command. The `function` command uses the basic following syntax:

```
> function.name <- function(INPUTS){YOUR ARGUMENTS}
```

Let's do a simple example. Let's say you wanted to make your own jitter function and call it "shaky":

```
> shaky <- function(x){x + rnorm(length(x))}
```

Now if we type:

```
> hmnrghts$jit.gnp <- shaky(hmnrghts$gnpcats)
```

Normal random errors will be added to each value of the `gnpcats` variable, and the resulting variable will be called "jit.gnp."

## 11 Maximum Likelihood in R

A nice feature of R's simple function definition is that R can readily use the full flexibility of Maximum Likelihood Estimation (MLE). If a "canned" model does not suit your research, and you want to derive your own estimator using MLE, then R can accommodate you. To illustrate how programming an MLE works in R, consider the estimator for the probability parameter in a binomial distribution. Our likelihood function is:

$$\pi^y(1 - \pi)^{n-y} \quad (1)$$

So our log-likelihood function is:

$$y \cdot \log(\pi) + (n - y) \log(1 - \pi) \quad (2)$$

We can easily define our log-likelihood function in R with the `function` command:

```
binomial.loglikelihood <- function(pi, y, n) {  
  loglikelihood <- y*log(pi) + (n-y)*log(1-pi)  
  return(loglikelihood)  
}
```

Now to get our estimate, we need R to find the value of  $\pi$  that maximizes the log-likelihood function given the data. We can do this with the `optim` command:

```
test <- optim(c(.5),           # starting value for pi  
  binomial.loglikelihood,     # the log-likelihood function  
  method="BFGS",             # optimization method  
  hessian=TRUE,               # return numerical Hessian  
  control=list(fnscale=-1),   # maximize function instead of minimize  
  y=43, n=100)                # the data  
print(test)
```

Note that `optim` is actually a minimizer, so we turn it into a maximizer with `fnscale=-1`.

A neat feature of defining our log-likelihood function is that we can readily call it to get a picture of the optimization problem:

```
ruler <- seq(0,1,0.01)  
loglikelihood <- binomial.loglikelihood(ruler, y=43, n=100)  
plot(ruler, loglikelihood, type="l", lwd=2, col="blue",  
  xlab="pi", ylab="L(pi)", ylim=c(-300,-70),  
  main="Log-Likelihood for Binomial Model")  
abline(v=.43)
```

## 12 Using Packages for Upper-Level Methods

In this section, we consider a few methods that go beyond the typical expectations for econometric software. To do this, we load a package and use some of the commands associated with it. Consider a hierarchical linear model: Mainline software is increasingly getting better at accommodating this framework, but most people have to use MLWin or HLM to run such models. In R, we need only load the `nlme` package to use such a model. Here is an example based on some data measured at UNC's Dental School:

```

library(nlme)
data(Orthodont)
fm1<-lme(distance~age, data=Orthodont) #everything is random
fm2<-lme(distance~age+Sex, data=Orthodont, random=~1) #only random intercept

```

For an example that is even more out-of-reach from most software, let's run a Bayesian regression model estimated with Markov Chain Monte Carlo (MCMC). We do this by installing `MCMCpack`, which was written by R users Andrew Martin and Kevin Quinn. To do this, we need to install two packages and load the library:

```

install.packages("MCMCpack")
library(MCMCpack)

```

Now we load a data set on Swiss demographics and run our Bayesian regression model:

```

data(swiss)
posterior1 <- MCMCregress(Fertility ~ Agriculture + Examination +
  Education + Catholic + Infant.Mortality, data=swiss)
summary(posterior1)

```

Just for fun, let's see how our Bayesian regression compared to classic OLS:

```

OLS1 <-lm(Fertility ~ Agriculture + Examination + Education +
  Catholic + Infant.Mortality, data=swiss)
summary(OLS1)

```

## 13 Object-Oriented Programming

As I mentioned before, R is an object-oriented environment. This means that you, as a researcher, have the opportunity to use sophisticated programming tools in your own research. In object-oriented programming, you create a “class” of item that has a variety of features. You then can create items within the class (called “objects” or “variables”) that will have unique features. In R, you can create classes from the object system `S3` or `S4`.

To illustrate, a “linear model” is a class of objects created by the `lm` command. It is an `S3` class. Earlier today, we created `hmnrghts.model`, which was an object of the linear model class. This object had features including `coefficients`, `residuals`, and `cov.unscaled`. In all `S3`-class objects, we can call a feature by writing the object name, the dollar sign, and the name of the feature we want. For example `hmnrghts.model$coefficients` would list the vector of coefficients from that model.

You may also define your own objects of either class. Here I offer an example of the more general `S4` class. These are excerpts from some code I have written, so they call functions I define elsewhere. To see the full version of the program, consult

[http://www.unc.edu/~monogan/research/FULL\\_SIMULATIONS.R](http://www.unc.edu/~monogan/research/FULL_SIMULATIONS.R). To start, any `S4` class needs to be defined with the `setClass` command, which defines the name of the class and all of its features, which are called “slots” for `S4`:

```

setClass('simulation',
  representation(outcomeA='matrix', outcomeD='matrix', bestResponseA='numeric',
    bestResponseD='numeric', equilibriumA='character', equilibriumD='character',
    parameters='character'))

```

Now I define a function that ends by returning an object of class “simulation”:

```

simulate<-function(type,v,delta,m.2,m.1=0.7,theta.A=seq(-1,1,.1),
  theta.D=seq(-1,1,.1)){
  #call the correct utility functions
  utilityA<-get(paste(type,"A",sep="."))
  utilityD<-get(paste(type,"D",sep="."))

  #create matrices and vectors
  outcomeA<-matrix(NA,21,21) #I need an error message if NA isn't replaced.
  outcomeD<-matrix(NA,21,21)
  bestResponseA<-rep(NA,21) #Again, NA needs to be replaced.
  bestResponseD<-rep(NA,21)
  equilibriumA<- 'NA' #NA may stick, but this is class 'character'
  equilibriumD<- 'NA'

  #matrix attributes
  rownames(outcomeA)<-seq(-1.0,1.0,0.1)
  colnames(outcomeA)<-seq(-1.0,1.0,0.1)
  rownames(outcomeD)<-seq(-1.0,1.0,0.1)
  colnames(outcomeD)<-seq(-1.0,1.0,0.1)
  names(bestResponseA)<-seq(-1.0,1.0,0.1)
  names(bestResponseD)<-seq(-1.0,1.0,0.1)

  #fill-in the utilities for all strategies for party A
  for (i in 1:21){
    for (j in 1:21){
      outcomeA[i,j]<-round(utilityA(m.1,m.2,v,delta,theta.A[i],
        theta.D[j]),4)
    }
  }

  #utilities for party D
  for (i in 1:21){
    for (j in 1:21){
      outcomeD[i,j]<-round(utilityD(m.1,m.2,v,delta,theta.A[i],
        theta.D[j]),4)
    }
  }

  #best responses for party A
  for (i in 1:21){
    bestResponseA[i]<-which.max(outcomeA[,i])
  }

  #best responses for party D
  for (i in 1:21){
    bestResponseD[i]<-which.max(outcomeD[i,])
  }

  #find the equilibria
  for (i in 1:21){

```

```

    if (bestResponseD[bestResponseA[i]]==i){
      equilibriumA<-dimnames(outcomeA)[[1]][bestResponseA[i]]
      equilibriumD<-dimnames(outcomeD)[[2]][bestResponseD[bestResponseA[i]]]
    }
  }

#save the input parameters in a slot that can be called
parms<-c(type,v,delta,m.2)

#call the simulation class and fill the slots
result<-new('simulation', outcomeA=outcomeA, outcomeD=outcomeD,
  bestResponseA=bestResponseA, bestResponseD=bestResponseD,
  equilibriumA=equilibriumA, equilibriumD=equilibriumD, parameters=parms)
result
}

```

Now that the function is defined, I can create an object called “treatment.1” and then print-out the value of the slot “equilibriumA” for this object. Notice that calling a feature is different in S4 because we now use the @ symbol instead of the \$ of S3:

```

treatment.1<-simulate("Quadratic",v=0.1,delta=0.0,m.2=-0.1)
treatment.1@equilibriumA

```

# Figures in R

## 14 The plot Function

For sections 14-19, we discuss graphs in the `base` package, which seem to be more popular due to their inclusion in Fox's *R and S-plus Companion*. Within the `base` framework, the `plot` function performs almost all graphics tasks. To see the total sum of arguments that one can call using `plot`, type `args(plot.default)`, which returns the following:

```
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
  log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
  ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
  panel.last = NULL, asp = NA, ...)
```

Obviously there is a lot going on underneath the generic `plot` function. But for the purpose of getting started with figure creation in R we want to ask what is essential. The answer is straightforward: one value `x` must be specified. Everything else has either a default value or is not essential. To start experimenting with `plot`, let's load data from John Fox's `car` package (`car = "Companion to Applied Regression"`). We're going to use Duncan's data on occupational prestige, appropriately titled "Prestige". The `summary` function gives some basic descriptive statistics of Duncan's prestige data.

```
> library(car)
> data(Duncan)
> attach(Duncan)
> summary(Duncan)
```

	type	income	education	prestige
bc	:21	Min. : 7.00	Min. : 7.00	Min. : 3.00
prof	:18	1st Qu.:21.00	1st Qu.: 26.00	1st Qu.:16.00
wc	: 6	Median :42.00	Median : 45.00	Median :41.00
		Mean :41.87	Mean : 52.56	Mean :47.69
		3rd Qu.:64.00	3rd Qu.: 84.00	3rd Qu.:81.00
		Max. :81.00	Max. :100.00	Max. :97.00

As a first foray into creating figures, we can plot the variables separately with the command `plot(varname)`.<sup>3</sup> For example, if we want to look at the distribution of the data points for `education`, we simply type `plot(education)` and Figure 2(a) is returned in the R graphics interface. Note that this figure plots `education` against a default index. Of course, we are more often interested in bivariate relationships. We can explore these easily by incorporating an `x` and a `y` in the call to `plot`: `plot(education, prestige)`. This produces Figure 2(b).

## 15 Figure Construction

R graphics are wonderful in that one begins with a blank slate. There are a good number of options that can be combined in a number of ways to create figures that suit your specific needs. These include scatterplots, line graphs, bar graphs, histograms, box plots, and more. It is perhaps

---

<sup>3</sup>For univariate plots, see section 19 on page 28 for better alternatives than `plot`. Two that you might find especially useful are the histogram function (`hist(varname)`) and the boxplot function (`boxplot(varname)`).

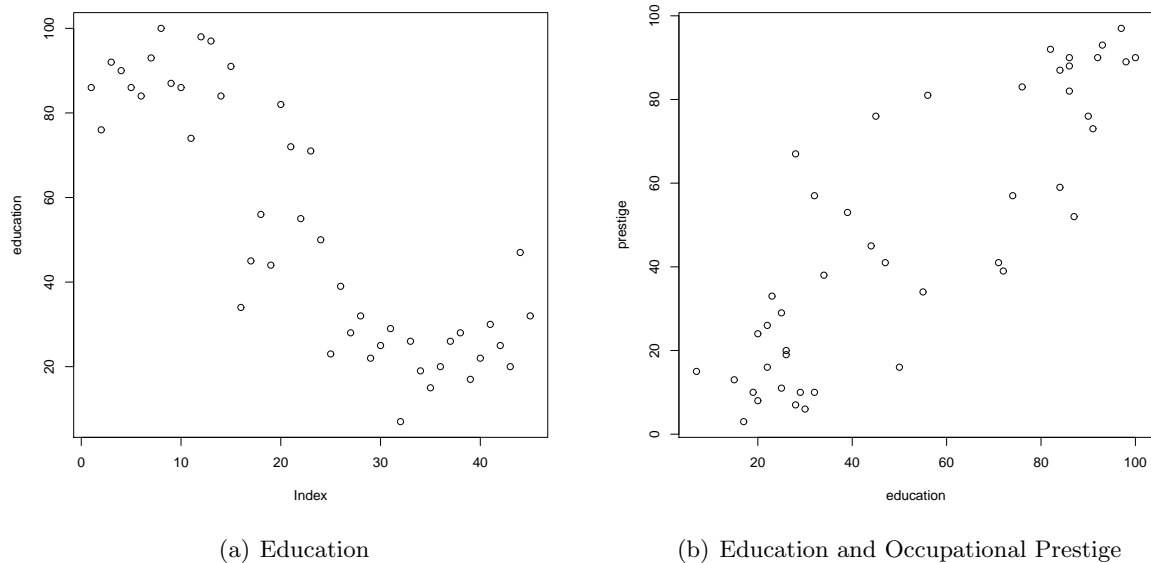


Figure 2: Education and Education by Occupational Prestige

most useful for getting started to work through the basic functions/options that bring considerable flexibility to creating figures in R.

**The Coordinate System:** In the above scatterplot, we were not worried about establishing the coordinate system because the data effectively did this for us. But often, you will want to establish the dimensions of the figure before plotting anything—especially if you are building up from the blank canvas. The most important point here is that your `x` and `y` must be of the same length. This is perhaps obvious, but missing data can create difficulties that will lead R to balk. One alternative is to establish the coordinate system outright by creating vectors like the following, which go into Figure 8:

```
> xaxis <- c(1:12)
> econ.inds <- c(2, 3, 3.5, 2, 3, 2.5, 3, 2.5, 3, 3.5, 4, 4)
> econ.reps <- econ.inds + 2
> econ.dems <- econ.inds - 1
```

We can now take these four vectors as data—that is, we can think of these four objects as variables. Let’s treat these data as four separate series, where `xaxis` represents some generic index for time and the other three objects represent average perceptions on some hypothetical scale for Independents, Republicans, and Democrats respectively.

**Plot Types:** We now want to plot these series, but the `plot` function allows for different types of plots. The different types that one can include within the generic `plot` function include:

- `type='p'` This is the default and it plots the `x` and `y` coordinates as *points*.
- `type='l'` This plots the `x` and `y` coordinates as *lines*.
- `type='n'` This plots the `x` and `y` coordinates as *nothing* (it sets up the coordinate space only).
- `type='o'` This plots the `x` and `y` coordinates as *points and lines* overlaid (i.e., it “over-plots”).

`type='h'` This plots the x and y coordinates as *histogram-like vertical lines*.

`type='s'` This plots the x and y coordinates as *stair-step like lines*.

**Axes:** It is possible to turn off the axes, to adjust the coordinate space by using the `xlim` and `ylim` options, and to create your own labels for the axes.

`axes=` Allows you to control whether the axes appear in the figure or not; I often like to turn them off by selecting `axes=F`. Then I create my own labels as follows:

- `axis(side=1, at=c(2, 4, 6, 8, 10, 12), labels=c("Feb", "Apr", "June", "Aug", "Oct", "Dec"))`

`xlim=, ylim=` For example, if we wanted to expand the space from the R default, we could enter:

- `plot(xaxis, econ.ind, type="o", xlim=c(-5, 17), ylim=c(-5, 15))`

`xlab="", ylab=""` Creates labels for the x- and y-axis (if you use this option you will want to be sure that you select `axes=F`).

**Style:** There are a number of options to adjust the style in the figure, including changes in the line type, line weight, color, point style, and more. Some that I commonly use include:

`lty=` Selects the type of line (solid, dashed, short-long dash, etc.)

`lwd=` Selects the line width (fat or skinny lines)

`pch=` Selects the plotting symbol, can either be a numbered symbol (`pch=1`) or a letter (`pch="R"`)

`col=` Selects the color of the lines/points in the figure (see “Color in R” for the many options)

**par** The `par` function brings added functionality to plotting in R. What is perhaps most important is that the `par` allows you to plot multiple (x, y)’s in a single graphic. This is accomplished by selecting `par(new=T)` following each call to `plot`.

## 15.1 Add-on Functions

There are also a number of add-on functions that one can use once the basic coordinate system has been created using `plot`. Some of these are listed below, and their implementation appears in the following sample code.

`arrows(x1, y1, x2, y2)` Create arrows within the plot (useful for labeling particular data points, series, etc)

`text(x1, x2, "text")` Create text within the plot (modify size of text using the character expansion option `cex`)

`lines(x, y)` Create a plot that connects lines

`points(x, y)` Create a plot of points

`polygon()` Create a polygon of any shape (rectangles, triangles, etc.)

`legend(x, y, at = c("", "",), labels=c("", ""))` Create a legend to identify the components in the figure

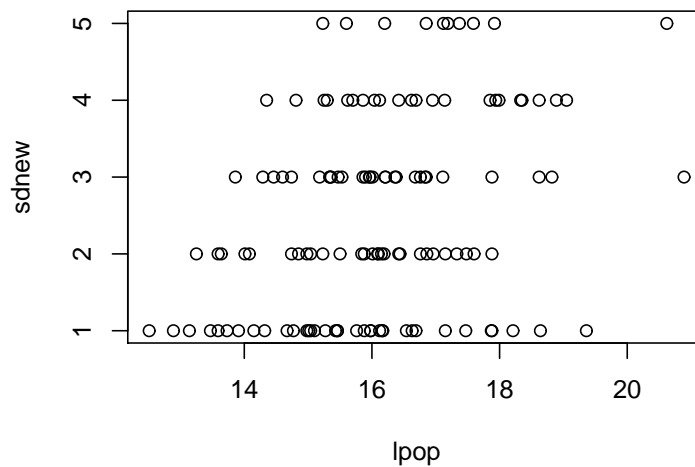


Figure 3: A Sample R Graphic

## 16 Regression Diagnostic Plots

To get a fuller sense of using scatterplots in R, let's run some pre- and post-regression diagnostics on our human rights data. You may need to re-run the following commands before plotting:

```
> detach(Duncan)
> hmnrights<-read.dta("D:/temp/hmnrights.dta")
> hmnrights.model<-lm(sdnew~democ+gnpcats, data=hmnrighs)
```

Now, let's say we wanted to plot human rights violations by the `lpop` variable. To do that we would use the following commands to get Figure 3:

```
> attach(hmnrights)
> plot(lpop, sdnew)
```

Given that `sdnew` is an ordinal variable it is harder to see where the preponderance of observations are. Often it helps to add some random perturbations to the plot. This is easily done in R with the `jitter` command, resulting in Figure 4:

```
> plot(lpop, jitter(sdnew))
```

Now let's use some of the subcommands for the `plot` function to add options to your graphics:

```
> plot(lpop, jitter(sdnew), xlab="Log of Population", ylab="Human
+ Rights Violations", main="Human Rights Violations by Population")
```

As seen in Figure 5, this adds labels for the x and y axis as well as a main title. A subtitle can be added as well.

We can also easily plot elements from the models that we estimated. Let's plot the fitted values from the regression model we estimated against the one of the regressors, the measure of population.

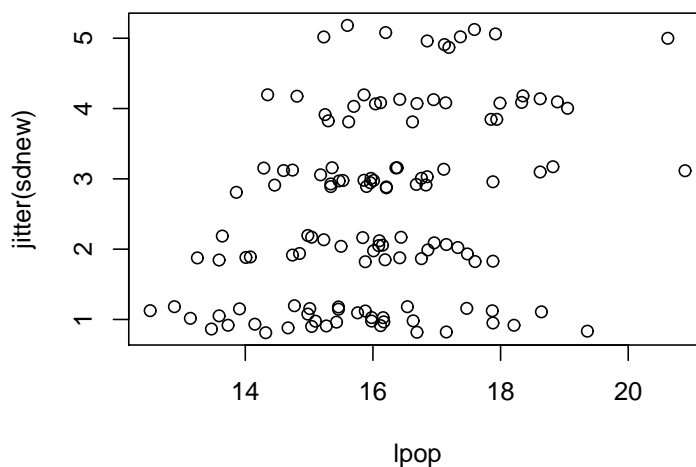


Figure 4: A Graphic with the Jitter Command

```
> plot(hmnrghs.model$fit, lpop, xlab='Fitted Human Rights Violations',
+ ylab='Population', main='Plot of Fitted Values')
```

In Figure 6, we can see the fitted values of our model against one of the regressors, a standard sight test for heteroscedasticity in linear models.

Let's say we wanted multiple graphs on a single page in order to make comparisons. For example, let's say we wanted to look at the residual plots for both population and GNP from the regression model we estimated earlier. To do that we have to use the `par` command. The `par` command is a lower level graphing command which allows you to make a variety of adjustments to graphs. Type `?par` to see all that it controls. Figure 7 is an example of a graph created with adjustments to the `par` command:

```
> par(mfrow=c(2,1))
> plot(lpop, hmnrghs.model$resid, ylab="OLS Residuals",
+ xlab="Population", main="Residual Plot")
> plot(jitter(as.numeric(gnpcats)), hmnrghs.model$resid, ylab="OLS Residuals",
+ xlab="GNP", main="")
> detach(hmnrghs)
```

Here the `par` command tells R to create a 2 by 1 set of graphs. You then need to supply two graphs. You can do up to eight graphs on a single page. By the look of Figure 7, it would appear that there is some linear relationship between GNP and the fitted values in our model, indicating heteroscedasticity.

**Graphic Output.** The easiest way to get graphics out of R is to simply right-click on the figure itself and then choose to save the figure as either a metafile (for use in Word) or as a postscript file (for use in  $\text{\LaTeX}$ ). The default size is for a full page graph. To resize the graph just use the mouse to resize the plot window. For  $\text{\LaTeX}$  users who want to set the bounding box size use the `postscript` command. Before you plot the graph, type:

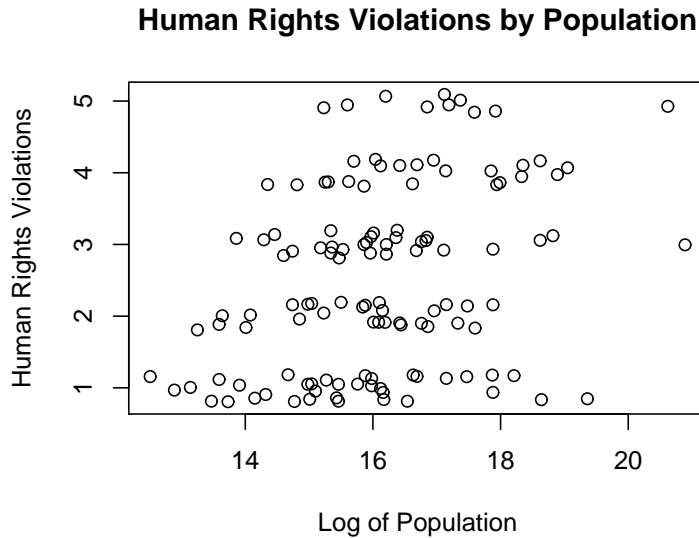


Figure 5: More R Graphics Commands

```
> postscript('FILENAME.eps', horizontal=FALSE, width=#, height=#).
```

This will set the width and height of the graphic in inches. You may have to experiment with this at first to get the size you want. The graphics in this document have a bounding box that is 5 inches wide and 4 inches high. The `horizontal` command changes the orientation of the graphic from landscape to portrait orientation on the page. Change it to `TRUE` to have the graphic adopt a landscape orientation. After you plot, type `dev.off()`.

Alternatively, for non- $\text{\LaTeX}$  users who would like to create graph output consistent with Microsoft Office software, a jpeg graph can be created with the commands:

```
> jpeg('C:/Temp/test.jpg')
> # INSERT GRAPHING COMMANDS
> dev.off()
```

Sections 17 and 18 offer example uses of these commands.

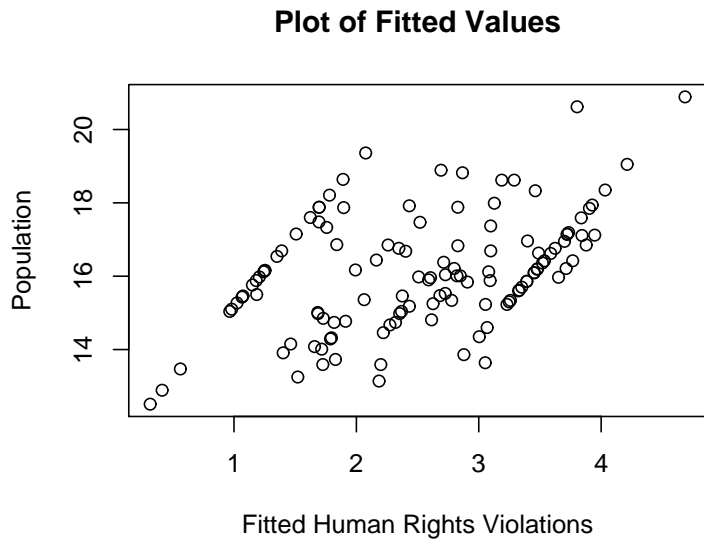


Figure 6: An OLS Fitted Values Plot

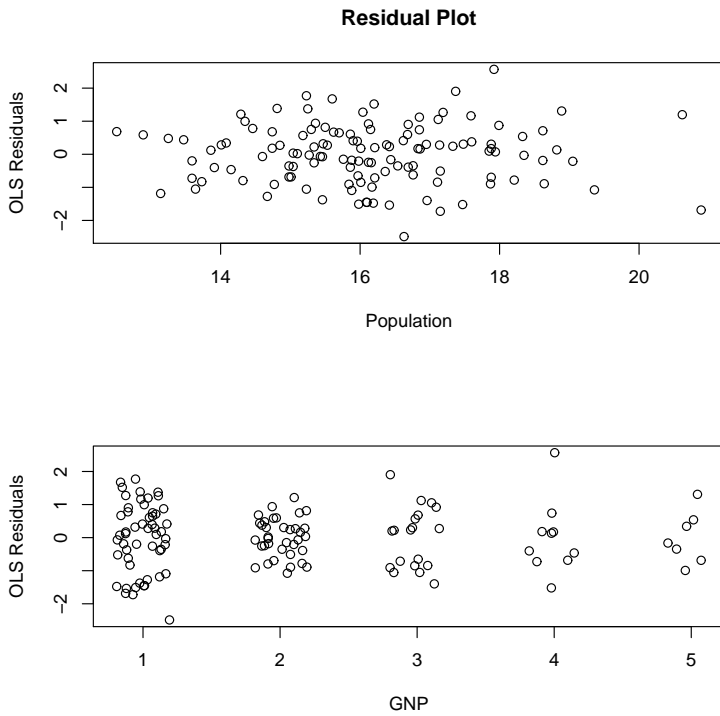


Figure 7: Including Multiple Graphs on the Same Page

## 17 Example Code: Time Series

```
#pdf("D:/temp/parallel.pdf")
#postscript("parallel.eps", horizontal=FALSE, width=7, height=7,
#          onefile=FALSE, paper="special", family="ComputerModern")
plot(xaxis, econ.ind, type="o", lwd=2, pch=5, ylim=c(0, 6.5),
     axes=F, xlab="", ylab="")
par(new=T)
plot(xaxis, econ.dems, type="o", lwd=2, pch=1, col="blue", ylim=c(0, 6.5),
     axes=F, xlab="", ylab="")
par(new=T)
plot(xaxis, econ.reps, type="o", lwd=2, pch=2, col="red", ylim=c(0, 6.5),
     axes=F, xlab="Month", ylab="Average Perception (Hypothetical)")
axis(side=1, at=c(2, 4, 6, 8, 10, 12),
     labels=c("Feb", "Apr", "June", "Aug", "Oct", "Dec"))
box()
text(2.25, 6.25, "Republicans", cex=1.2)
text(6, 3.7, "Independents", cex=1.2)
text(10.5, 1.2, "Democrats", cex=1.2)
arrows(2, 6.05, 3, 5.65, length=.10, lwd=2)
arrows(5.75, 3.5, 6.75, 3.1, length=.10, lwd=2)
arrows(10.25, 1.35, 9.25, 1.75, length=.10, lwd=2)
# dev.off()
```

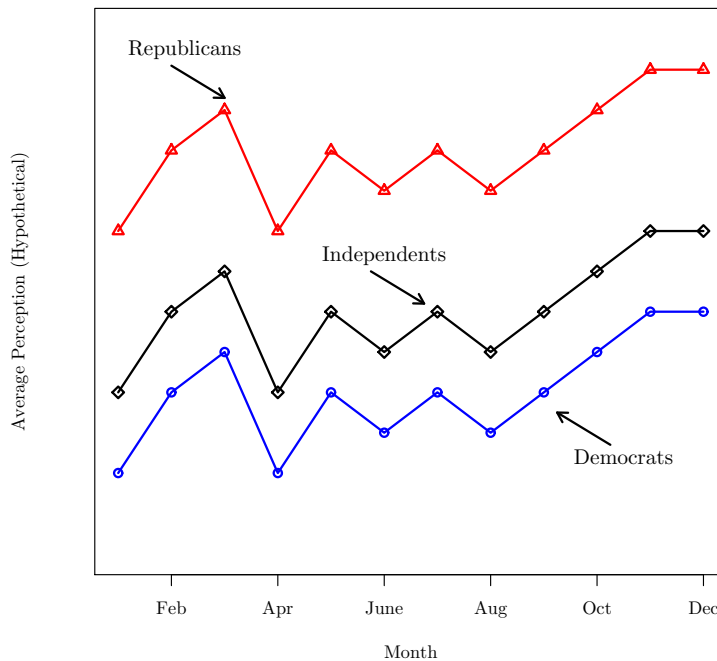


Figure 8: Hypothetical Example of Objective Information Updating

## 18 Example Code: Bar Graph

```
#jpeg("D:/temp/folpa.jpeg")
#postscript("folpa.eps", horizontal=FALSE, width=7, height=7,
# onefile=FALSE, paper="special", family="ComputerModern")
plot(c(0, 4), c(0, 0.35), type='n', xlab="How Interested in Politics?",
      ylab="Proportion", axes=F)
axis(2)
axis(side = 1, at = c(.5, 1.5, 2.5, 3.5),
      labels = c("Not at All", "Hardly", "Quite", "Very"))
polygon(c(0.25, 0.25, .75, .75), c(0, .33, .33, 0), col="gray76")
polygon(c(1.25, 1.25, 1.75, 1.75), c(0, .20, .20, 0), col="gray56")
polygon(c(2.25, 2.25, 2.75, 2.75), c(0, .32, .32, 0), col="gray36")
polygon(c(3.25, 3.25, 3.75, 3.75), c(0, .16, .16, 0), col="gray16")
abline(h=0, col="gray70")
title("Distribution of Political Interest")
text(.5, .35, ".33", cex=1.2)
text(1.5, .22, ".20", cex=1.2)
text(2.5, .34, ".32", cex=1.2)
text(3.5, .18, ".16", cex=1.2)
box()
# dev.off()
```

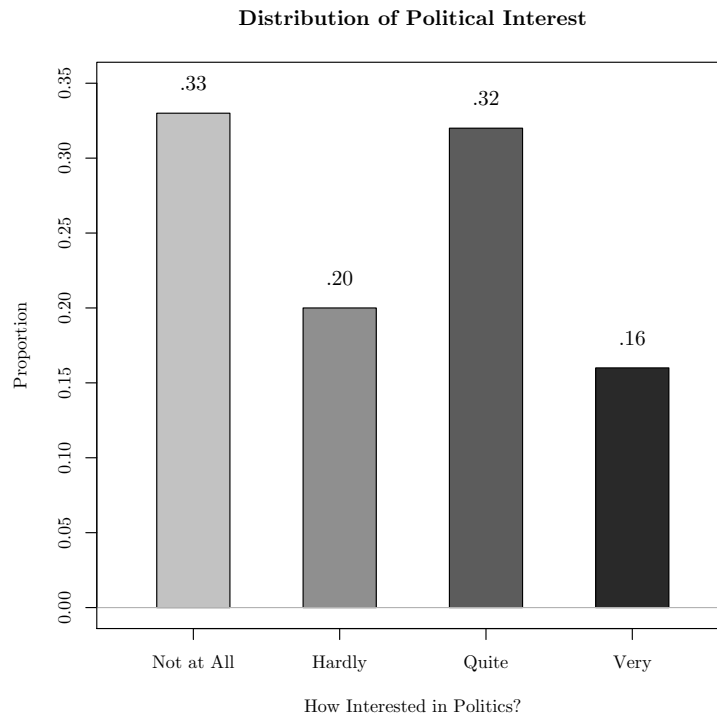


Figure 9: Attentiveness to Politics in the Albanian Mass Public

## 19 Non-plot Functions from base

There are a few graphing commands in the `base` package that do not call the `plot` function. One is the `boxplot` command, which we try using our data from UNC's dental school:

```
library(nlme)
data(Orthodont)
attach(Orthodont)
boxplot(distance~age)
```

The other command is `hist`, which gives us a histogram of a variable:

```
hist(distance)
box()
hist(distance,breaks=c(15,20,25,30,35))
detach(Orthodont)
```

## 20 Using lattice Graphics in R

As an alternative to the `base` graphics package, you may want to consider the `lattice` add-on package. These produce `trellis` graphics from the S language, which tend to make better displays of grouped data and numerous observations. A few nice features of the `lattice` package are that the graphs have viewer-friendly defaults and the commands do not require you to `attach` the data beforehand. We will simply touch on these graphs today, but for more information type `?xyplot` into the R command prompt or see Bill Jacoby's website at Michigan State, <http://polisci.msu.edu/jacoby/icpsr/graphics/>.

To start, the following code loads the library, runs a simple scatterplot, and then applies some of the options available to the command:

```
library(lattice)
xyplot(distance~age, data=Orthodont)
xyplot(distance~jitter(age), data=Orthodont, col="black")
```

We also might like to draw a graph that is similar to a simple scatterplot, but uses ordinal statistics for the vertical axis variable:

```
stripplot(distance~age, data=Orthodont)
dotplot(distance~age, data=Orthodont)
```

Lastly, if we want to get a good look at the distribution of a variable, we can plot the density or histogram of the variable:

```
densityplot(~distance, data=Orthodont)
histogram(~distance, data=Orthodont)
histogram(~distance | Sex, data=Orthodont)
```